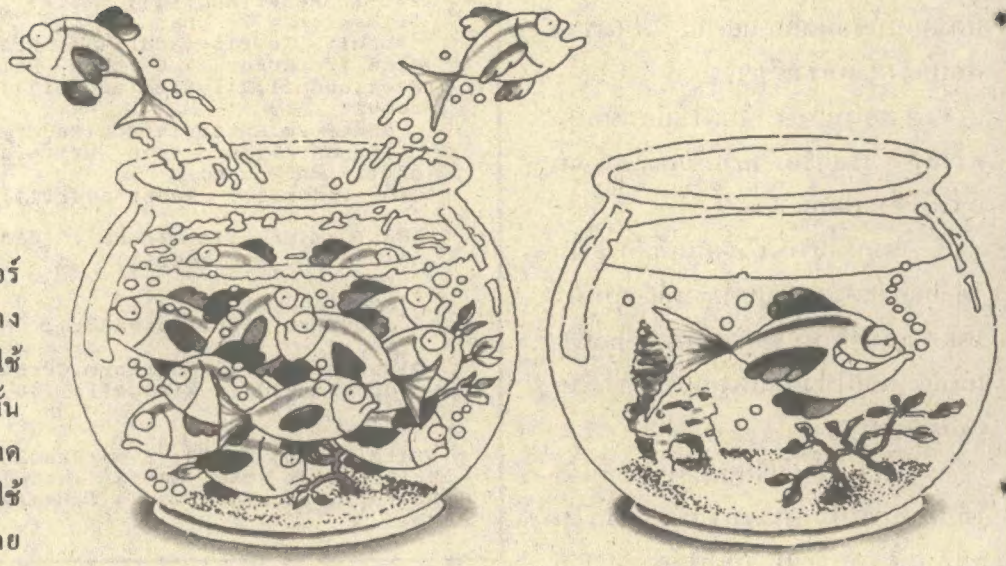
ชัยยง รัตนาไพศาลสุข

SOFT MAIL

DATA COMPRESSION
เครื่องมือย่อขนาดข้อมูล

ช่องทางที่ช่วยให้
การโอนย้ายข้อมูล
ทำได้ง่ายดายยิ่ง
โดยไม่วิตกเรื่องขนาด



ในปัจจุบันมีการนำคอมพิวเตอร์มาใช้ในงานด้านต่าง ๆ อย่างแพร่หลาย ทำให้ขนาดของข้อมูลที่ใช้ในปัจจุบันนี้ใหญ่โตกว่าสมัยก่อน ดังนั้นจึงมีกรรมวิธีต่าง ๆ ช่วยในการลดขนาดของข้อมูล ทั้งนี้เพื่อประหยัดเนื้อที่ที่ใช้เก็บข้อมูล และยังทำให้การโอนย้าย (tranfer) ข้อมูลระหว่างเครื่องคอมพิวเตอร์เป็นไปได้อย่างรวดเร็ว เราเรียกว่า Data Compression นั่นเอง ในที่นี้ผู้เขียนจะขอกล่าวถึงอัลกอริทึมที่น่าสนใจ และควรนำมาศึกษาเพื่อพัฒนาต่อไปในเรื่องของการทำ Data Compression โดยใช้หลักการของฮัฟฟ์แมนอัลกอริทึม (Huffman Algorithm)

เริ่มด้วยการตีความจากปัญหา

ก่อนที่จะเข้าไปในรายละเอียดของฮัฟฟ์แมนอัลกอริทึม ผู้เขียนจะชี้แจงภาพรวม ๆ ของวิธีเก็บข้อมูลภายในเครื่องคอมพิวเตอร์ เพื่อให้ผู้อ่านสามารถเข้าใจหลักการของฮัฟฟ์แมนอัลกอริทึมที่จะกล่าวถึงในภายหลังได้เป็นอย่างดี

เนื่องจากตัวอักษรภาษาอังกฤษ (Text File) มีลักษณะการเก็บในเครื่องคอมพิวเตอร์โดยการแทนด้วยรหัสต่างๆ เช่น เอ็บซีดิกหรือแอสกี เป็นต้น แต่ในบทความนี้จะมุ่งเน้นเกี่ยวกับการเก็บเท็กซ์ไฟล์ภายในเครื่องพีซีคอมพิวเตอร์เท่านั้น ซึ่งเครื่องเหล่านี้จะเก็บตัวอักษรต่าง ๆ ด้วยรหัสแอสกี 8 บิต ขนาดของตัวอักษร 1 ตัวจะใช้รหัสไบนารีจำนวน 8 บิตแทนได้ และกินเนื้อที่ในการเก็บเท่ากับ 1 ไบต์ จัดเป็นความยาวที่แน่นอนแปรเปลี่ยนไม่ได้ (Fix Length Code) ดังนั้นถ้าขนาดของเท็กซ์ไฟล์ใหญ่โตกว่าเดิม ย่อมจำเป็นต้องใช้เนื้อที่ในการเก็บมากขึ้นไปด้วย โดยขนาดของเนื้อที่จะขึ้นอยู่กับจำนวนของตัวอักษรนั่นเอง หากเราสามารถกำหนดเนื้อที่ที่ใช้เก็บตัวอักษรแต่ละตัวให้มีขนาดเปลี่ยนแปลงได้ตามความเหมาะสม เช่น ประโยคที่ว่า "For those of you who remain skeptical about the ability of this method to shorten files. Let's take an example the sentence you are now reading" ประโยคนี้ถ้าเก็บด้วยวิธีปกติจะใช้เนื้อที่จำนวน 147 ไบต์หรือเท่ากับ 147×8 = 1176 บิต ลองพิจารณาตัวอักษรต่าง ๆ ในประโยคนี้ เราจะพบว่า ประโยคดังกล่าวมีตัวอักษรที่แตกต่างกันเพียง 27 ตัวเท่านั้น เราสามารถแทนตัวอักษรแต่ละตัวด้วยรหัสขนาด 5 บิตก็เป็นการเพียงพอแล้ว เพราะว่ามันสามารถกระจายตัวอักษรที่แตกต่างกันได้มากถึง 2^5 = 32 ตัว เราจะใช้เนื้อที่ในการเก็บเพียง 147×5 = 715 บิต ซึ่งวิธีนี้เป็นวิธีที่มีประสิทธิภาพพอควร แต่ยังคงใช้ลักษณะการเก็บแบบ Fix Length Code เหมือนเดิม ถ้าเรา

สามารถเก็บตัวอักษรแต่ละตัวด้วยขนาดของบิตที่ไม่เท่ากันได้ (Variable Length Code) จะช่วยลดขนาดของเนื้อที่เก็บข้อมูลได้อีกมาก วิธีที่จะใช้พิจารณาว่าตัวอักษรใดควรเก็บด้วยจำนวนบิตเท่าไรนั้น เราสามารถใช้ความถี่ของตัวอักษรที่ปรากฏอยู่ในประโยคมาเป็นหลักเกณฑ์ในการพิจารณา โดยที่ ถ้าหากว่าประโยคนั้นมีความถี่ของตัวอักษรหลาย ๆ ตัว เราก็นำตัวอักษรนั้นมาแทนด้วยไบนารีโค้ดขนาดสั้น ๆ ส่วนตัวอักษรใดปรากฏน้อยครั้ง เราก็แทนด้วยไบนารีโค้ดที่มีขนาดยาวขึ้นหน่อย จากนั้นก็นำไบนารีโค้ดที่มีความยาวแตกต่างกันตามความถี่ของตัวอักษรมาเรียงต่อ ๆ กัน ทำให้ประหยัดเนื้อที่ที่ใช้เก็บข้อมูลได้มากกว่า ซึ่งวิธีการนี้คือแนวความคิดสำคัญของฮัฟฟ์แมนอัลกอริทึมนั่นเอง จากการใช้หลักการนี้ทดสอบกับประโยคตัวอย่างข้างต้นแล้วพบว่า ขนาดของข้อมูลที่ใช้ในการเก็บประโยคดังกล่าวนี้จะเหลือเพียง 612 บิต (เท่ากับ 612/8 ประมาณ 77 ไบต์) เท่านั้น ผู้อ่านลองพิจารณาการแทนตัวอักษรในประโยคดังกล่าวที่ได้แสดงไว้ในตารางที่ 1

คาแรกเตอร์	Frequency	รหัส	คาแรกเตอร์	Frequency	รหัส
space	26	101	m	3	111011
e	17	010	w	2	111111
t	13	1000	k	2	111110
o	12	1001	p	2	111100
a	10	1101	b	2	111101
s	8	0111	d	2	011001
i	7	0000	c	2	011000
h	7	0001	x	1	1110101
n	7	0010	g	1	1110100
l	5	1101	F	1	1110010
r	5	11000	' (apostrophe)	1	1110011
f	3	01101	, (comma)	1	1110001
y	3	00111	. (period)	1	1110000
u	3	00110			

ตารางที่ 1. แสดงความถี่ของตัวอักษรพร้อมรหัส

หลังจากเราได้ทราบแนวความคิดของการเก็บข้อมูลแบบ Variable Length Code เราจะพบว่ามีปัญหาใหญ่ ๆ เกิดขึ้น 3 ประการคือ

1. Variable Length Code สามารถทำการ Compress Data ได้ก็จริง แต่ในระบบคอมพิวเตอร์จะยอมรับเฉพาะการทำงานแบบ Fix Length Code ที่มีขนาด 8 บิตเท่านั้น เราอาจจะแก้ไขได้โดยการนำบิตของแต่ละตัวอักษรมาเรียงต่อ ๆ กัน เมื่อครบ 8 บิต ก็แทนด้วยตัวอักษร 1 ตัว ทำเช่นนี้เรื่อยไปจนข้อมูลหมด

2. การเก็บข้อมูลแบบ Variable Length Code อาจทำลายข้อมูลทั้งหมดได้ง่าย เพราะการนำบิตมาเรียงต่อกัน ถ้าเกิดการ encode ผิดเพียงบิตเดียวจะส่งผลให้การ Decompress ให้ข้อมูลกลับมาอยู่ในสภาพเดิมไม่สำเร็จ ข้อมูลที่ได้จะขาดความถูกต้อง ซึ่งจุดนี้เป็นเรื่องที่ควรระมัดระวังมากที่สุดในการทำ Data Compression

3. ปัญหาในการแยกแยะความแตกต่างของข้อมูลแต่ละบิตออกจากกันขณะทำการ Decompression เช่น เราสมมติให้ตัวอักษร A แทนด้วยรหัส 0, ตัวอักษร B แทนด้วยรหัส 1 และตัวอักษร C แทนด้วยรหัส 01 ในการทำ Decompress แล้วพบรหัส 01 เราไม่สามารถรู้ได้เลยว่ารหัสนี้แทนตัวอักษร AB ติดกัน หรือตัวอักษร C เพียงตัวเดียว ในการแก้ปัญหานี้เราต้องใช้ Prefix Property ของ Tree (Prefix Property คือต้องไม่มีรหัสตัวใดที่จะเป็น Prefix ของลำดับรหัสอื่น ๆ) เช่น ในตัวอย่างข้างต้นจะเห็นได้ว่า รหัสของตัวอักษร A เป็น Prefix ของตัวอักษร C ดังนั้นการแทนตัวอักษรด้วยรหัสดังกล่าวจะไม่มีคุณสมบัติของ Prefix แต่ในตารางที่ 1 การแทนตัวอักษรทุกตัวจะใช้คุณสมบัติของ Prefix Property ดังนั้นในการ Decompression จะไม่เกิดปัญหาที่กล่าวมา

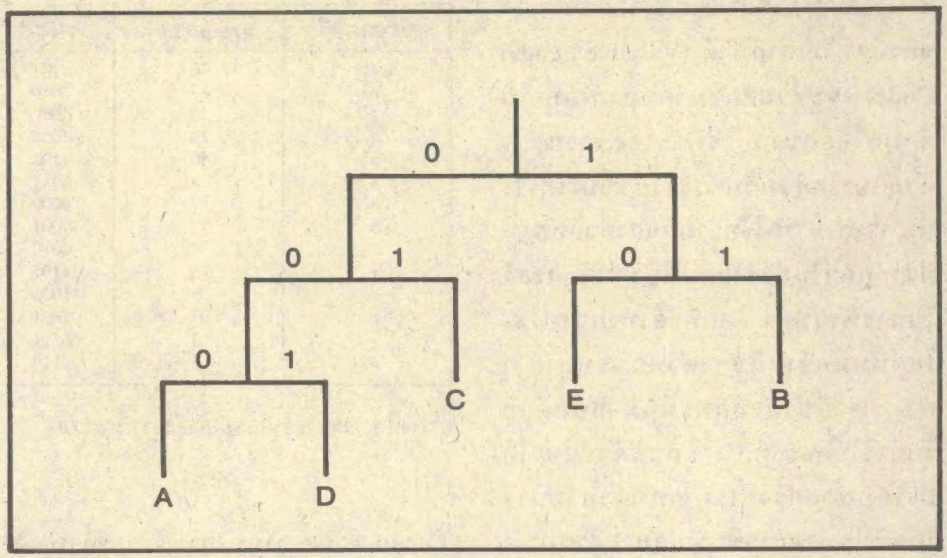
นานาทรรศนะว่าด้วยฮัฟฟ์แมนอัลกอริทึม

ถ้าเราพิจารณา Frequency Table ในตารางที่ 1 ซึ่งการ encode อักษรแต่ละตัวจะมีคุณสมบัติ Prefix Property ดังที่ได้กล่าวมาแล้ว เราจะพบว่ารหัสที่มีคุณสมบัติ Prefix Property นั้นจะสามารถแทนได้ด้วยไบนารีทรี ซึ่งก็คือ Tree ที่มี Children อยู่เพียง 2 Children ในแต่ละโหนดเท่านั้น โดยเราให้ Left Child แทนด้วยรหัส 0 และ Right Child แทนด้วยรหัส 1 ลองพิจารณาการแทนตัวอักษรในรูปที่ 1

ในรูปที่ 1 เราสามารถแทนตัวอักษรด้วยไบนารีโค้ดโดยการ trace จาก Root ไปยัง Leaf หากเรา trace ไปทาง Left Branch เราก็จะได้ '0' เป็นส่วนหนึ่งของรหัส และถ้า trace ไปทาง Right Branch จะได้ '1' เป็นส่วนหนึ่งของรหัส ด้วยวิธีการนี้เราจะพบว่าไม่มีตัวอักษรใดเลยที่เป็น Prefix ของตัวอักษรอื่น เพราะตัวอักษรทุกตัวจะปรากฏอยู่ที่ Leaf เท่านั้น

ต่อไปเราจะพิจารณาถึงการสร้างไบนารีทรีที่จะใช้เป็นส่วนหนึ่งในการทำ Data Compression โดยดูจากอัลกอริทึมในรูปที่ 2 ซึ่งเริ่มต้นจากการนำคาแรกเตอร์ที่ปรากฏใน Frequency Table มาบรรจุลงใน List ของ Subtree แล้วพิจารณาหาคาแรกเตอร์ที่มีความถี่น้อยที่สุดจำนวน 2 ตัว อาจจะมีความถี่เท่ากันหรือไม่ก็ได้ แล้วนำโหนดทั้งสองนี้มาผนวกเข้าด้วยกันให้เป็น Subtree โดยมี Children เป็นคาแรกเตอร์ของทั้งคู่ assign ผลรวมความถี่ของทั้ง 2 โหนดให้เป็นความถี่ของ Subtree จากนั้นนำ Subtree ที่ได้บรรจุกลับไปยัง List ของ Subtree ต่อจากนั้นให้พิจารณาหา Subtree ที่มีความถี่น้อยที่สุดจำนวน 2 Subtree แล้วนำมาทำการสร้าง Subtree ขึ้นมาใหม่ โดยยึดหลักการเดิม อัลกอริทึมนี้จะผนวก Subtree จนกระทั่งเหลือ Subtree ที่ใหญ่ที่สุดเพียงชุดเดียว หรือเรียกว่า Huffman Code Tree นั่นเอง (ดูรูปที่ 3 ประกอบ)

ข้อน่าสังเกต ฮัฟฟ์แมนอัลกอริทึมไม่มีหลักการที่แน่นอนในการผนวก Subtrees เข้าด้วยกัน หรือบ่งบอกว่าควรจะเอา Subtree ใดเป็น Leftchild และ/หรือ Subtree ใดเป็น Right Child ดังนั้นการ compress ด้วยวิธีนี้จึงอาศัยความถี่ของตัวอักษร แต่บางกรณีที่ถ้าเรา compress เป็น Word โดยใช้อัลกอริทึมเดิม เราจะสามารถลดขนาดของข้อมูลได้มากขึ้นกว่าการ compress เป็นตัวอักษร ตัวอย่างเช่น ประโยค "John where Bill had had had had had had had" เราแทนคำว่า had ด้วย 1, แทนคำว่า Bill ด้วย 01, แทน John ด้วย 000 และแทน where ด้วย 001 เราจะพบว่าขนาดของข้อมูลจะเล็กลงไปอีก

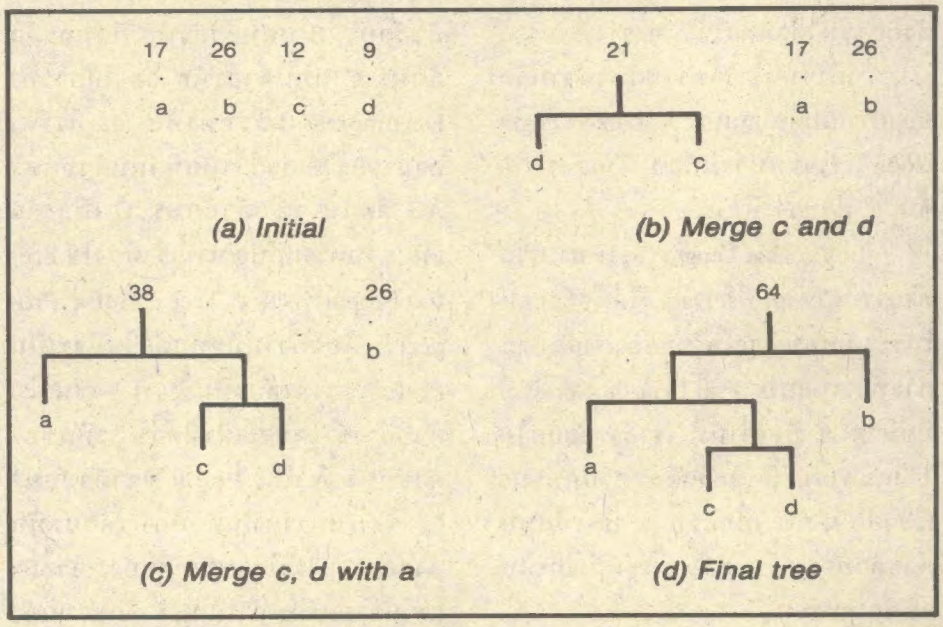


รูปที่ 1. แสดง Simple Code Tree

HUFFMAN ALGORITHM

For each character with a nonzero frequency
add the character to the list of subtrees
While the list of subtrees contains more than one subtree;
remove from the list the two subtrees
with the smallest frequencies;
make them the children of a new subtree whose frequency
is the sum of their frequencies;
add the new subtree to the list;
The remaining subtree is the Huffman code tree

รูปที่ 2. แสดงฮัฟฟ์แมนอัลกอริทึม



รูปที่ 3 แสดงขั้นตอนในโครงร่างของ Huffman Tree

ทว่าในความเป็นจริง เราไม่สามารถทำได้ดังตัวอย่างที่ยกมา เพราะปกติแล้วการเก็บไบนารีโค้ดต่างๆในคอมพิวเตอร์จะใช้รหัสขนาด 8 บิต ซึ่งสามารถทำการ represent ค่าที่แตกต่างกันได้ 256 ค่า ฉะนั้นถ้าหากเรา represent Word ด้วยรหัสดังกล่าว เราจะได้ Word ที่แตกต่างกันอย่างมากที่สุดเพียง 256 Words ไม่เพียงพอที่จะใช้ในการทำงานจริง และถึงแม้ว่าเราจะพยายามฝืนใช้หลักการนี้ เราย่อมพบว่าเราจำเป็นต้องแทน Word ต่าง ๆ ด้วยรหัสที่มีขนาดมากกว่า 8 บิตอย่างแน่นอน ทำให้ขนาดของข้อมูลที่ผ่านการ compress แล้วใหญ่โตขึ้น ซึ่งผิดจากแนวความคิดของการ Compression ที่ต้องการให้ขนาดของข้อมูลเล็กลง

ส่วนสนับสนุนในฮัฟฟ์แมน

จากนี้เป็นต้นไปเราจะมาพูดถึงการ Implement ฮัฟฟ์แมนอัลกอริทึมให้เป็นโปรแกรมที่สามารถทำงานได้จริง สิ่งแรกที่ต้องทำคือ เราต้องมี FOREST (หมายถึง List ของ Subtree ที่จะนำมา combine กัน) โดยที่ FOREST จะเก็บ Weight (ความถี่) และพอยน์เตอร์ที่ชี้ไปยัง Subtree นั้น นอกจากนี้เรายังต้องสร้าง TREE ขึ้นเพื่อใช้ในการเก็บ Huffman Code Tree และสิ่งที่ขาดไม่ได้คือเราจะต้องสร้าง ALPHABET เพื่อใช้ในการเก็บตัวอักษร และพอยน์เตอร์ที่ชี้มายังตัวของมันเองให้เป็นประโยชน์ในการอ้างอิงภายหลัง

ดังนั้นโปรแกรมที่ผู้เขียนสร้างขึ้นจึงประกอบด้วยเรคอร์ดหลัก ๆ 3 เรคอร์ดด้วยกันคือ

1. TREE เรคอร์ดของ Tree ซึ่งจะเป็นตัวเก็บ Tree ที่จะมีขนาดโตขึ้นเรื่อย ๆ จากการผนวก Subtree จนในที่สุดจะได้ Huffman Code Tree

2. ALPHABET เรคอร์ดของ Alphabet ซึ่งจะใช้ในการอ้างอิงต่อไป

3. FOREST เรคอร์ดของ Forest จะเป็นตัวที่ใช้เก็บ List ของ Subtrees

การทำงานของโปรแกรมจะเริ่มต้นจากการอ่านซอร์สไฟล์ที่ต้องการ compress โดยอ่านเข้ามาทีละคาแรกเตอร์ แล้วสร้าง Frequency Table ขึ้นมา ซึ่งตารางนี้ก็คืออะเรย์ของความถี่ที่สามารถอินเด็กซ์ได้ด้วยตัวอักษรนั้น หลังจากนั้นเราจะนำอะเรย์ของ Frequency มาบรรจุลงในอะเรย์ของ ALPHABET เพื่อใช้อ้างอิงในภายหลัง และบรรจุลงในอะเรย์ของ FOREST ซึ่งการบรรจุลงในอะเรย์ของ FOREST นี้ก็เป็นการ initial ค่าโหนดเริ่มต้น (Leaf) นั่นเอง

โปรแกรมจะทำการหาโหนดที่น้อยที่สุดจำนวน 2 โหนดมา combine กัน โดยจะสร้าง Subtree ขึ้นมาใหม่ให้มี Children เป็น 2 โหนด โดยให้มี Leftchild เป็นโหนดที่มี Weight (ความถี่) น้อยกว่าโหนดที่เป็น Rightchild แล้ว assign Weight ให้กับ Subtree เสียใหม่ โดยการนำเอา Weight จากโหนดทั้งสองมารวมกัน แล้วบรรจุ Subtree ใหม่นี้กลับคืนไปให้ FOREST

โปรแกรมจะสร้าง Subtree ที่มีลักษณะเหมือน Subtree ที่บรรจุใน FOREST ลงใน TREE ด้วย จากนั้นโปรแกรมจะทำงานในลักษณะนี้ตั้งแต่ต้นจนจบ เมื่อ Subtree ถูก combine หมดจะได้ Huffman Code Tree ลองพิจารณาโครงร่างการทำงานอย่างคร่าวๆ ในรูปที่ 4, 5 และ 6

```
(1) While there is more than one tree in the forest do
    begin
(2)  i:=index of the tree in FOREST with smallest weight;
(3)  J:=index of the tree in FOREST with second smallest weight
(4)  create a new node with left child FOREST[i].root and
        right child FOREST[j].root;
(5)  replace tree i in FOREST by a tree whose root is the new
        node and whose weight is
        FOREST[i].weight+FOREST[j].weight;
(6)  delete tree j from FOREST
    end;
```

รูปที่ 4 แสดงโครงร่างคร่าว ๆ ของ Huffman tree construction

รูปที่ 5 แสดงโปรซีเดอร์ทั้งสอง

```
procedure lightones(var least,second:integer);
 { sets least and second to the indices in FOREST of
   the trees of smallest weight. We assume lasttree >= 2 }
 var
  i : integer;
begin {initialize least and second, considering first two
       trees}
  if FOREST[1].weight <= FOREST[2].weight then
```

รูปที่ 5 (ต่อ)

```
        begin least := 1; second := 2  end
    else
        begin least := 2; second := 1  end;
    { Now let i run from 3 to lasttree. At each iteration least
      is the tree of smallest weight among the first  i trees
      in FOREST ,and second is the next smallest of these }
    for i := 3 to lasttree do
       if FOREST[i].weight < FOREST[least].weight then
          begin second := least; least := i  end
       else if FOREST[i].weight < FOREST[second].weight then
            second := i
    end { lightones }

function create (lefttree,righttree : integer) : integer;
 { returns new node whose left and right children are
   FOREST[lefttree].root and FOREST[righttree].root }
 begin
   lastnode := lastnode + 1;
       { cell for new node is TREE[lastnode] }
   TREE[lastnode].leftchild := FOREST[lefttree].root;
   TREE[lastnode.rightchild := FOREST[righttree].root;
   { now enter parent pointers for new node and its children }
   TREE[lastnode].parent := 0;
   TREE[FOREST[lefttree].root].parent := lastnode;
   TREE[FOREST[righttree].root].parent := lastnode;
   return (lastnode)
 end; { create }
```

```
precedure Huffman;
  var
    i,j: integer; { the two trees of least weight in FOREST }
    newroot: integer;
  begin
    while lasttree > 1 do begin
       lightones(i,j);
       newroot := create(i,j);
       { Now replace tree i by the tree whose root is newroot }
        FOREST[i].weight := FOREST[i].weight+FOREST[j].weight;
        FOREST[i].root := newroot;
       { next, replace tree j, which is no longer needed, by
        lasttree, and shrink FOREST by one }
        FOREST[j] := FOREST[lasttree];
        lasttree := lasttree - 1;
    end
  end; { Huffman }
```

รูปที่ 6 แสดงอัลกอริทึมของฮัฟฟ์แมน

รูปที่ 7 และ 8 แสดงโครงสร้างข้อมูลหลังจากการทำงานตามอัลกอริทึมของฮัฟฟ์แมนผ่านไปสองรอบ

จากโปรซีเดอร์ข้างต้นผู้เขียนใช้อินทีเจอร์แทนการใช้พอยน์เตอร์ ทั้งนี้เพื่อให้ผู้อ่านเกิดความเข้าใจได้ง่ายขึ้น

ทดลองจากโปรแกรมตัวอย่าง

รายละเอียดของโปรแกรมที่ผู้เขียนได้เขียนขึ้นเพื่อทำการ Compression นั้นจำเป็นต้องมีโปรแกรมหลัก 2 โปรแกรมคือ โปรแกรม compress ทำหน้าที่ encoding และโปรแกรม decompress หรือ decoding โดยที่อินพุตของโปรแกรม Encoding จะเป็นเท็กซ์ไฟล์ทั่ว ๆ ไป เมื่อผ่านการ compress แล้วจะได้เอาต์พุตเป็น Encoded ไฟล์ ซึ่งจะต้องนำมาใช้เป็นอินพุตของโปรแกรม Decoding ในภายหลัง

ความยุ่งยากของโปรแกรมก็คือเราจะต้องเสาะหาวิธีเก็บ Huffman Code Tree ลงในดิสค์ให้ได้ในรูปของรหัสไบนารี 8 บิต เพื่อใช้ในการ decode ต่อไป

ผู้อ่านลองพิจารณาอัลกอริทึมในรูปที่ 9 ที่ใช้ในการ Write Tree ให้อยู่ในรูปของไบนารีโค้ดเพื่อการบันทึกลงดิสค์ได้

สร้างโปรแกรมเข้ารหัส

โปรแกรมจะเริ่มต้นจากการอ่านเท็กซ์ไฟล์แล้วทำการสร้าง Frequency Table ซึ่งเป็นอะเรย์ของ Integer Indexed ที่แสดงด้วยคาแรกเตอร์ พร้อมทั้งเก็บจำนวนตัวอักษรทั้งหมดที่อยู่ในเท็กซ์ไฟล์นี้ไว้ เพื่อจะได้ใช้งานในการ Decode ส่วน Frequency Table จะนำไปใช้ในการสร้าง Huffman Code Tree ตามอัลกอริทึมของฮัฟฟ์แมน หลังจากนั้น โปรแกรมจะทำการ represent แต่ละคาแรกเตอร์ด้วยรหัสที่ได้จากการ travel ไปใน Branch ต่าง ๆ โดยโปรแกรมจะเริ่มต้น travel ที่ Root ออกไปยัง Left Branch และ Right Branch ตามลำดับ หากเป็นการ travel ไปทางซ้าย โปรแกรมจะให้เอาต์พุตเป็น 0 ถ้าโปรแกรม travel ไปทางขวาจะให้เอาต์พุตเป็น 1 วิธีนี้เราจะได้รหัสที่มีขนาดบิตต่างกัน สามารถใช้แทนตัวอักษรต่าง ๆ ได้ เราก็จะเก็บรหัสของตัวอักษรเหล่านี้ไว้ในอะเรย์ เพื่อใช้ในการ encode จากนั้นโปรแกรมจะทำการเปิดเอาต์พุตไฟล์ เริ่มต้นด้วยการบรรจุจำนวนตัวอักษรทั้งหมดที่อยู่ในเท็กซ์ไฟล์ที่ได้เก็บไว้โดยใช้เนื้อที่จำนวน 2 ไบต์ สามารถเก็บขนาดของอินพุตไฟล์ที่ใหญ่สุดได้ถึง 65535 ไบต์ โปรแกรมจะทำการเก็บ Huffman Code Tree ที่ผ่านอัลกอริทึม Writetree แล้ว ซึ่งจะมีรูปแบบเป็นคาแรกเตอร์หลาย ๆ ตัวเรียงกัน โดยจำนวนคาแรกเตอร์จะขึ้นอยู่กับขนาดของ Huffman Code Tree นั่นเอง

เมื่อเสร็จสิ้นการ write Heading ของเอาต์พุตแล้วจะเป็นการ write ข้อมูลจริง ซึ่งโปรแกรมจะเริ่มอ่านอินพุตไฟล์ (เท็กซ์ไฟล์) อีกครั้งทีละตัวอักษร แล้วนำมาเปลี่ยนเป็นรหัสที่ได้เตรียมไว้ แล้วนำรหัสเหล่านั้นมาเรียงต่อกัน (อย่าลืมว่ารหัสเหล่านี้อยู่ในรูปของบิต) เมื่อครบ 8 บิตจะทำการ write ลงเอาต์พุตไฟล์ โปรแกรมจะทำงานในลักษณะนี้จนกว่าตัวอักษรในอินพุตไฟล์จะหมด เราก็จะได้ไฟล์ที่ผ่านการ compress แล้ว

	Weight	Root
0	12	0
1	40	1
2	15	2
3	8	3
4	25	4

FOREST

	Symbol	Freq.	Leaf
0	a	12	0
1	b	40	1
2	c	15	2
3	d	8	3
4	e	25	4

ALPHABET

	Left Child	Right Child	Parent
0	−1	−1	−1
1	−1	−1	−1
2	−1	−1	−1
3	−1	−1	−1
4	−1	−1	−1

TREE

รูปที่ 7 แสดง Initial Contents ของอะเรย์

	Weight	Root
0	25	4
1	40	1
2	35	6

FOREST

	Symbol	Freq.	Leaf
0	a	12	0
1	b	40	1
2	c	15	2
3	d	8	3
4	e	25	4

ALPHABET

	Left Child	Right Child	Parent
0	−1	−1	5
1	−1	−1	−1
2	−1	−1	6
3	−1	−1	5
4	−1	−1	−1
5	3	0	6
6	2	5	−1

TREE

รูปที่ 8. แสดงโครงสร้างข้อมูลแบบต้นไม้หลังจากทำงานผ่านไปสองรอบ

Algorithm Writetree โดยใช้ Tree node เป็นอินพุต
ถ้าโหนดที่เข้ามาเป็นลีฟ
ให้เอาต์พุตเป็น 0 แล้วตามด้วยรหัส 8 บิตของตัวอักษรที่อยู่ที่ลีฟนั้น
แต่ถ้าโหนดที่เข้ามาไม่ใช่ลีฟ
ให้เอาต์พุตเป็น 1
ทำการ Recursive อัลกอริทึมนี้ โดยใช้ Leftchild ในการผ่านค่า
ทำการ Recursive อัลกอริทึมนี้ โดยใช้ Rightchild ในการผ่านค่า

รูปที่ 9. แสดง Writetree อัลกอริทึม

ถอดรหัสโปรแกรม

วิธีการถอดรหัสจะเริ่มต้นด้วยการอ่านจำนวนเท็กซ์ไฟล์จากตัวอักษร 2 ตัว (ซึ่งก็คือ 2 ไบต์แรกใน Heading ของ Compressed ไฟล์) จากนั้นโปรแกรมจะอ่านตัวอักษรต่อไป แล้วนำมา converse กลับอีกครั้งให้เป็น Huffman Code Tree สำหรับขั้นตอนในการ reconstruct Huffman Code Tree จากตัวอักษรที่อ่านเข้ามา ผู้เขียนได้แสดงเป็นอัลกอริทึมประกอบการพิจารณาไว้ในรูปที่ 10

เมื่อเราจัดการ reconstruct เสร็จเรียบร้อย โปรแกรมจะอ่านบิตถัดไปซึ่งก็คือส่วนหนึ่งของรหัสแทนตัวอักษรที่ compress ไว้แล้ว โปรแกรมจะคอยตรวจสอบว่าบิตที่อ่านเข้ามาเป็น 0 หรือ 1 ถ้าเป็น 0 โปรแกรมจะทำการ travel ไปทางซ้าย หรือ travel ไปทางขวาเมื่อบิตที่ได้เป็น 1 โปรแกรมจะอ่านและตรวจสอบเช่นนี้ไปเรื่อยๆจนกระทั่งถึง Leaf ของ Tree นั้น เพื่อให้ได้ตัวอักษรออกมา 1 ตัว โปรแกรมจะทำงานในลักษณะเดิม พร้อมทั้งนับจำนวนตัวอักษรที่ได้ออกมาว่ามีค่าเท่ากับเท็กซ์ไฟล์ที่เก็บไว้ตั้งแต่แรกหรือยัง

```
Algorithm Readtree จะทำการ Return Tree
อ่านค่าบิตจาก file;
ถ้าหากบิตที่อ่านเข้ามาเป็น 1
        ทำการ Recursive Readtree แล้วให้ค่าเก็บไว้ใน Left;
        ทำการ Recursive Readtree แล้วให้ค่าเก็บไว้ใน Right;
        สร้าง Tree ขึ้นมาโดยให้ Leftchild และ Rightchild เป็น Left และ Right ตามลำดับ;
        ให้ Symbol ของ Tree นั้นเป็น Null character (00H);
        Return (Tree)
ถ้าหากบิตที่อ่านเข้ามาเป็น 0
        จะทำการอ่านค่า 8 บิตถัดไปแล้วนำมาแปลงกลับเป็นตัวอักษร;
        สร้าง Tree ขึ้นมาโดยให้ Leftchild และ Rightchild เป็น NULL;
        ให้ Symbol ของ Tree เป็นตัวอักษรตัวที่ได้จากการแปลง;
        Return (Tree);
```

รูปที่ 10 แสดง Readtree อัลกอริทึม

อุปสรรคย่อมเกิดขึ้นได้

ผู้เขียนประสบปัญหาบางประการที่เกิดขึ้นกับโปรแกรมตัวอย่าง กล่าวคือ โดยปกติเมื่อทำการ compress ไฟล์แล้วจะได้ไบนารีโค้ดขึ้นมา แต่เราไม่สามารถล่วงรู้ได้เลยว่า ไบนารีโค้ดนี้ใช้แอสกีโค้ดอะไร ซึ่งในการทำงานจริงย่อมมีโอกาสที่จะได้รหัส End-of-file ดังนั้นในการ decompress เมื่อโปรแกรมอ่านตัวอักษรแล้วพบรหัสจบแฟ้ม จะทำให้โปรแกรมเกิดการ terminate การ Decompress จึงไม่สมบูรณ์

ผู้เขียนได้แก้ไขปัญหาที่เกิดขึ้นนี้โดยพิจารณาจากแนวความคิดเดิม นั่นคือ โปรแกรมจะจัดการ encode หรือ decode กับข้อมูลที่มีจำนวนครบ 8 บิตเท่านั้น บิตต่าง ๆ เหล่านี้ล้วนเกิดจาก Information ของอินพุตดาต้าทั้งสิ้น แต่ถ้าเราดัดแปลงแนวความคิดนี้ โดยการให้ encode หรือ decode กับบิตที่มีขนาดเพียง 7 บิต แล้วทำการ initial ค่าให้กับบิตซ้ายสุดด้วย 1 ดังนั้นช่วงของ Compressed Character ที่ได้จะอยู่ในช่วงระหว่าง 80H จนถึง 255H (10000000 ถึง 11111111) ด้วยวิธีการนี้จะทำให้ Encoded Character ไม่มีโอกาสเป็นรหัส End-of-file เลย ทั้งนี้เนื่องจากรหัส End-of-file มีค่าเป็น 1AH (00011010) วิธีการนี้จะทำให้ Compressed ไฟล์มีขนาดใหญ่โตขึ้นกว่าเดิมอีกเล็กน้อย ส่วนหลักการทำงานของการ Compress และ Decompress โปรแกรมยังคงใช้หลักการเดิมอยู่

บทสรุป

เนื่องจากผู้เขียนมีเวลาค่อนข้างจำกัด ดังนั้นโปรแกรมที่เขียนขึ้นจะใช้เวลาในการทำงานค่อนข้างมากถ้าอินพุตไฟล์มีขนาดใหญ่ และสามารถ compress ได้เฉพาะเท็กซ์ไฟล์ที่เป็นภาษาอังกฤษเท่านั้น โปรแกรมนี้เขียนด้วยภาษาซีอย่างง่าย ๆ (คอมพายล์โดยใช้ Quick C Compiler เวอร์ชัน 1.0) ทั้งนี้เพื่อแสดงให้เห็นถึงหลักของการ Compression โดยใช้อัลกอริทึมของฮัฟฟ์แมน ซึ่งเป็นหัวข้อหนึ่งในวิชาโครงสร้างข้อมูล ผู้อ่านที่สนใจอัลกอริทึมนี้ ลองศึกษาเพิ่มเติมได้จากคู่มือ Data Structure ทั่ว ๆ ไป

เอกสารอ้างอิง

1. Alfred V.Aho, John E.Hopcroft and Jeffrey D.Ullman: "Data Structures and algorithms".
2. Werner Feibel: "Advanced Quick C", McGraw-Hill, 1988.
3. Amstredam J. : "Data Compression with Huffman Coding", Programming Project, BYTE, May 1986.
4. Arron M.Tenenbaum, Moshe J.Augenstein: "Data Structures using PASCAL", Prentice/Hall international, 1981.